Problem 1. Dining Gourmands

a)

MODULE Gourmands =
CONST N := 6
VAR utensil: 0 .. N-1 -> Mutex
% utensil[2k] are knives
% utensil[2k+1] are forks

FUNC inc(n: 0 .. N-1) = n = N-1 => RET 0
[*] RET n+1

THREAD Gourmand(id: 0 .. N-1) =
VAR knife, fork: 0 .. N-1, eaten : Int := 0 |
IF x // 2 = 0 =>
  knife := id;
  fork := inc(id);
[*]
  fork := id;
  knife := inc(id);
FI
DO
% P1
  utensil(knife).acq;
% P2
  utensil(fork).acq;
% P3
  eaten := eaten + 1;
% P4
  << utensil(knife).rel; utensil(fork).rel >>
OD

b) Deadlocks cannot happen in this system. We show that at every point in time one of the 6 threads can make progress.

First observe that assignment of fork and knife variables is correct given the indices of gourmands. Next observe that if there exists a thread at point P3 or point P4 these threads can always make progress.

If a thread T is waiting at point P2 for a fork, then the neighbor thread T' owning that fork must be at point P3 or P4, so T' can make progress. Finally, if a thread T is waiting at point P1 for a knife, then the thread T'' owning that knife is at position P2, P3 or P4, so it either T'' make progress or the other neighbor T''' ≠ T is at P3 or P4 and can thus make progress.

We can also observe that dependencies between threads are paths of length at most three containing neighboring threads and such paths cannot be cyclic because there is no resource that can cause the last thread wait on the first thread in the path.
Problem 2. FIFO Queues

a)

MODULE BoundBufferImpl[T] EXPORT Produce, Consume =
CONST N := 1024
TYPE BI = 0 .. N-1
VAR b : BI -> T
  first: BI := 0
  last: BI := 0
  m: Mutex := m.new();
  cc: Condition := c.new();
  cp: Condition := c.new();

FUNC inc(bi: BI) -> BI =
  bi = N-1 => RET 0
  [*] RET (bi+1)

FUNC bufferFull() -> Bool = RET (inc(last)=first)

FUNC bufferEmpty() -> Bool = RET (last=first)

PROC push(t) =
  b(last) := t;
  last := inc(last);

PROC pop() -> T =
  VAR t := b(first);
  first := inc(first);
  RET t

PROC Produce(t) =
  m.acq;
  DO bufferFull() => cp.wait(m) OD;
  IF bufferEmpty() => cc.broadcast [*] SKIP FI;
  push(t);
  m.rel;

PROC Consume() -> T = VAR t |
  m.acq;
  DO bufferEmpty() => cc.wait(m) OD;
  IF bufferFull() => cp.broadcast [*] SKIP FI
  t := pop();
  m.rel;
  RET t;

Note that Produce and Consume invoke broadcast before the buffer is brought to the state where the threads waiting can execute. There is no problem with this because we use the semantics of monitors as given on page 14 of the Handout 17. According to this semantics, broadcast (as well as signal) operations move the threads waiting on the condition variable into the queue for the lock m but the lock remains to be held by the procedure executing broadcast (or signal). It is therefore only after m.rel that procedures woken up can continue the execution (if it happens to be the one that gets the lock).

b) We can prove that BoundBufferImpl implements Buffer using an abstraction function. The abstraction function maps a state with \( k \) threads in BoundBufferImpl into the state with \( k \) threads in Buffer.
The buffer content in `BoundBufferImpl` is mapped to `Buffer.b` through function `F`:

```plaintext
FUNC F(b: (BI -> T), first: BI, last: BI) -> SEQ T =
IF (first=last) => RET {}
[*] { b(first) } + F(b,inc(first),last)
FI
```

Let the labelling of program points in the implementation be the following:

PROC Produce(t) =
[PB]  m.acq;
    DO bufferFull() => cp.wait(m) OD;
[PCB]  IF bufferEmpty() => cc.broadcast [*] SKIP FI;
        push(t);
        m.rel;
[PCCE]
PROC Consume() -> T = VAR t |
[CB]  m.acq;
    DO bufferEmpty() => cc.wait(m) OD;
[CCB]  IF bufferFull() => cp.broadcast [*] SKIP FI
        t := pop();
        m.rel;
[CCCE]  RET t;
```

It is a property of the Mutex specification on page 11 of Handout 17 (and can be proved once and for all) that two different threads cannot be simultaneously in the region enclosed by `m.acq` and `m.rel` statements. Hence `m.acq` and `m.rel` define a critical region. Moreover, once a thread enters the critical region it can continue to make progress in that critical region and other threads are prevented from entering the critical region. (See page 7 of Handout 17.)

By examining the Condition specification, we observe that `broadcast` (as well as `signal`) do not release the lock at all. On the other hand, `wait` releases the lock temporarily, but reacquires it by the time it returns. From this we conclude that program points between PCB and PCE and between CCB and CCE form a critical region. Next, we observe that the critical region protects the buffer in the appropriate way:

1. Buffer is not accessed from outside the critical region. In this case, buffer data structure is implemented in `BoundBufferImpl` with variables `b.first` and `last`. The only place where these variables are accessed is `push` and `pop`, both of which are called only from within the critical region.

2. The code in the critical region only reads and writes variables that cannot be accessed by another thread outside the critical region. In this case, the only variables accessed are `b.first, last`, as well as the local variable `p` that is inaccessible from outside the thread.

These two conditions are sufficient to guarantee that we may transform the critical region into an atomic regions without increasing the set of external traces. The result is the following.

PROC Produce(t) =
[PB]  m.acq;
    DO bufferFull() => cp.wait(m) OD;
[PCB]  IF bufferEmpty() => cc.broadcast [*] SKIP FI;
        push(t);
        m.rel; >>
[PCCE]
PROC Consume() -> T = VAR t |
[CB]  m.acq;
    DO bufferEmpty() => cc.wait(m) OD;
We can now essentially ignore the code in Produce except for \texttt{push} procedure call and ignore the code in Consume except for \texttt{pop} procedure call, because the other parts of the code do not change the image of the state under the function $F$. Function $F$ specifies the mapping of the buffer variables. We just need to specify the mapping of program counters. This mapping has the same form for every thread. Let the labelling of program points in the specification be the following:

\begin{verbatim}
MODULE Buffer[T] EXPORT Produce, Consume =
VAR b : SEQ T := {}
APROC Produce(t) = [P1] << b + := {t} >> [P2]
APROC Consume() -> T = VAR t | [C1] << b # {} => t := b.head; b := b.tail; RET t >> [C2]
END Buffer
\end{verbatim}

We map the state at [PB] in Produce to [P1]. All program points between [PB] and [PCB] as well as [PCB] itself is also mapped to [P1]. Implementation states with program counter PCE are mapped to states with program counter [P2]. We proceed similarly for mapping of program points in Consume: points from [CB] up to and including [CCB] are mapped to [C1] whereas point CCE and the point after it are mapped to [C2].

For the simulation relation, we need to show that the initial state of the buffer data structure is mapped to the empty sequence, and that for every external step in the implementation there exists a corresponding step in the specification. The base case is easy because initially

$$\text{first} = \text{last} = 0$$

so $F$ returns $\emptyset$.

For the initial and final actions for the invocation of Produce and Consume in the implementation, there exist corresponding initial and final actions in the specification.

For the atomic action from [PCB] to [PCE] in BoundBufferImpl the corresponding action is the invocation of the body of Buffer.Produce. Let the initial state in BoundBufferImpl have values

$$\text{first, last, b}$$

After executing Produce the state components have values

$$\text{first, inc(last), b\{last->t\}}$$

So we need to show that

$$F(\text{first, last, b} + \{t\} = F(\text{first, inc(last), b\{last->t\}}) \quad (1)$$

But observe that that if the test for bufferFull is true, then the loop exits without releasing the lock, so in those cases bufferFull test can be thought of as belonging to the atomic region from [PCB] to [PCE]. So \texttt{~bufferFull()} holds, which means that it is \textit{not} the case that

$$\text{last} = \text{first} + N - 1 \pmod N$$

Therefore, $\text{last} = (\text{first} + k) \pmod N$ where $0 \leq k \leq N - 2$. By induction on $k$ we can now prove the equation 1. Assume

$$F(\text{first, last, b} + \{t\} = F(\text{first, inc(last), b\{last->t\}})$$

Then
F(first,inc(last),b) + {t}
= (by definition of F since ~(inc(last)=first) )
{b(first)} + F(inc(first),inc(last),b) + {t}
= (by induction hypothesis)
{b(first)} + F(inc(first),inc(last)),b{inc(last)->t}),
= (by definition of F since ~(inc(last)=inc(first)) )
F(first,inc(last)),b{inc(last)->t})

Showing that the atomic action of BoundBufferImpl.Consume corresponds to the body of Buffer.Consume is in our implementation analogous to the previous proof for BoundBufferImpl.Produce so we omit it.

For all steps other than procedure initiations, terminations, atomic action from [PCB] to [PCE] and atomic action from [CCB] to [CCE], the abstract state does not change. With these actions we can associate empty sequence of transitions in the specification since they have no external labels.

This finishes the sketch of the proof that BoundBufferImpl implements Buffer.

c) We assume that the scheduler has the following property: it never produces an infinite trace with the property that: an action inside Produce or Consume remains enabled infinitely many times, but never executes.

From this it follows that if one of the Produce actions is at point [PCB] or a subsequent point, then this Produce action will complete. Similarly if some Consume action is at point [CCB] or later, then that action will eventually complete. Also, if a Produce procedure is at a point before [PCB] and not waiting on cp then at some later point it will either go through the loop and terminate or it will end up waiting at cp. Similarly Consume action either terminates or ends up waiting on cc.

To show that some action eventually completes, assume the opposite: after some time, Produce and Consume stop completing even though both Produce and Consume keep initiating. This means that both Produce and Consume actions follow the path that keeps them within the DO loop, otherwise they would terminate. Since the procedures remain in their loops, the state of the buffer does not change. Since new actions keep being initiated and (by fairness) entering the loop, it means that both bufferFull() and bufferEmpty() return true, even though the buffer does not change at all. This is clearly impossible since the same buffer of positive capacity cannot be both full and empty. We arrived at a contradiction, which means that the assumption that no action completes is wrong.

Problem 3. Locks and Data Structures

a)

  |   |    |
  /\   |
  |   |

Deadlock occurs when some instance A of deleteOne procedure acquires lock on node [3] by a call to traverse, and then another instance B acquires a lock on node [2] by a call to traverse. Next A attempts to acquire lock on [2] and waits for B which in turn attempts to acquire lock on [3] and waits for A.

b)


Deadlocks cannot occur because the list is initially acyclic and remains acyclic. When deleteOne operation acquires two locks, it always acquires first lock on the element which is earlier in the list.
c) According to b), it is sufficient to guarantee acyclicity of the list:

\[
Acyclic(first, next) = \neg (\exists s: SEQ Nodes \mid \\
\quad s.\text{head} = first \land \\
\quad \forall i: 0 \ldots s.\text{size}-2 \mid \\
\quad \quad next(s(i)) = s(i+1) \land \\
\quad \quad \exists k: 0 \ldots s.\text{size}-2 \mid \\
\quad \quad \quad s(s.\text{size}-1) = s(k))
\]

We observe that this condition is also necessary: if there is a cycle of length \(k\) then \(k\) invocations of \texttt{deleteOne}(i) can cause a deadlock (for suitably chosen values of the parameters \(i\)).

d) Property \((P)\) does not hold. The following scenario illustrates the problem.

Let thread \(A\) be executing \texttt{deleteOne}(3) and let the initial list contain elements \([0,1,2,3,4]\). Let the second execution of \texttt{advance} set \(n=[2]\) and \(t=[3]\). Consider the program point \(PX\) in \texttt{advance}.

\[
\text{PROC advance(n) =} \\
\quad t = next(n); \text{m(n).rel}; % [PX] \\
\quad \text{m(t).acq; RET t}
\]

At this point thread \(A\) holds no locks and the state is the following:

\[
\begin{align*}
0 & \rightarrow 1 & \rightarrow 2 & \rightarrow 3 & \rightarrow 4 & \rightarrow \text{null} \\
| & | & | & | & | \\
| & | & | & | & | \\
\end{align*}
\]

\[
| n \quad t
\]

Assume that second thread \(B\) comes in and calls \texttt{deleteOne}(2) and performs deletion. The result is the following:

\[
\begin{align*}
\text{-------------} \\
\text{|} & | \\
\text{|} & | \\
\text{| \textbackslash/} & | \\
\text{|} & | \\
\text{| \textbackslash/} & | \\
\text{|} & | \\
\end{align*}
\]

\[
\begin{align*}
| & | & | & | & | \\
| & | & | & | & | \\
\end{align*}
\]

\[
| n \quad t
\]

After \texttt{deleteOne}(2) in thread \(B\) finishes, it removes all its locks and thread \(A\) resumes its \texttt{deleteOne}(2) operation. After \texttt{deleteOne}(2) completes, the result is:

\[
\begin{align*}
\text{-------------} \\
\text{|} & | \\
\text{|} & | \\
\text{| \textbackslash/} & | \\
\text{|} & | \\
\text{| \textbackslash/} & | \\
\text{|} & | \\
\end{align*}
\]

\[
\begin{align*}
| & | & | & | & | \\
| & | & | & | & | \\
\end{align*}
\]

\[
| n \quad \text{-------------}
\]

So the result after two calls to \texttt{deleteOne} is the list \([0, 1, 2, 4]\). This means that only one element was removed from the list.

We can prevent this scenario if we use \textit{lock coupling} while traversing the list in \texttt{advance}:
PROC advance(n) =
    t = next(n); m(t).acq; m(n).rel; RET t

This ensures that after successful completion of \( k \) calls to \texttt{deleteOne} (in a long list) the set of elements reachable from \texttt{first} has \( k \) elements fewer than initially. The reason is roughly that \texttt{deleteOne} operations cannot “overtake” one another since each operation keeps a lock on an object before the place where modification happens. In fact, it can be shown that the result of concurrent execution is the same as if \texttt{deleteOne} operations executed sequentially in the same order in which they initiated the concurrent execution.

Deadlocks cannot occur either because among the \texttt{deleteOne} operations executing the one that initiated earliest can always make progress. (Recall we are assuming an acyclic list.)